Risk-Based and Functional Security Testing

C. C. Michael, Cigital, Inc. [vita[3]]
Will Radosevich, Cigital, Inc. [vita[4]]

2005-09-23; Updated 2009-07-23 by Ken van Wyk [vita[5]]

L3/L4 / L, M[6]

This article discusses the role of software testing in a security-oriented software development process. It focuses on two related topics: functional security testing and risk-based security testing. Functional testing is meant to ensure that software behaves as it should. Therefore, it is largely based on software requirements. Risk-based testing is based on software risks, and each test is intended to probe a specific risk that was previously identified through risk analysis.

Introduction

This document discusses the role of software testing in a security-oriented software development process. It focuses on two related topics: functional security testing and risk-based security testing.

Functional testing is meant to ensure that software behaves as it should. Therefore, it is largely based on software requirements. For example, if security requirements state that the length of any user input must be checked, then functional testing is part of the process of determining whether this requirement was implemented and whether it works correctly.

Analogously, risk-based testing is based on software risks, and each test is intended to probe a specific risk that was previously identified through risk analysis. A simple example is that in many web-based applications, there is a risk of injection attacks, where an attacker fools the server into displaying results of arbitrary SQL queries. A risk-based test might actually try to carry out an injection attack, or at least provide evidence that such an attack is possible. For a more complex example, consider the case where risk analysis determines that there are ambiguous requirements. In this case, testers must determine how the ambiguous

3. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/251-BSI.html (Michael, C. C.)
4. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/252-BSI.html (Radosevich, Will)
7. #dsy255-BSI_introduction
8. #dsy255-BSI_buscase
9. #dsy255-BSI_sstest
10. #dsy255-BSI_functest
11. #dsy255-BSI_risk-based-testing
12. #dsy255-BSI_sectest
13. #dsy255-BSI_sectestactiv
14. #dsy255-BSI_metrics
15. #dsy255-BSI_case-study
16. #dsy255-BSI_glossary

requirements might manifest themselves as vulnerabilities. The actual tests are then aimed at probing those vulnerabilities.[21]

This document focuses on how risk-based and functional security testing mesh into the software development process. Many aspects of software testing are discussed, especially in their relationship to security testing. Nonetheless, this document is not intended as a primer on software testing per se. Test engineers should be familiar with standard references for software testing, such as [Binder 99[22]], [Dustin 99[23]], [Fewster 99[24]], [Marick 94[25]], [Beizer 95[26]], [Black 02[27]], and [Kaner 99[28]].

Business Case for Security Testing

From a technical and project management perspective, security test activities are primarily performed to validate a system's conformance to security requirements and to identify potential security vulnerabilities within the system. From a business perspective, security test activities are often conducted to reduce overall project costs, protect an organization's reputation or brand, reduce litigation expenses, or conform to regulatory requirements.

Identifying and addressing software security vulnerabilities prior to product deployment assists in accomplishing these business goals. Security test activities are one method used to identify and address security vulnerabilities. Unfortunately, in some organizations, this is the only method used to identify security vulnerabilities.

Understanding the high-level benefits of security test activities is relatively easy to understand. However, obtaining the real cost benefit of security test activities has historically been a difficult task.

The software testing and quality assurance community has done an excellent job of identifying the cost benefits of conducting tests to identify software bugs early and often. If one considers that security vulnerabilities are also a form of software bugs, the same conclusions can be made for security testing. In parallel with the QA community, the security industry and law enforcement community have been compiling statistics on the cost of security incidents over the last ten years. The information gathered by both of these communities is helpful to understanding the business case for security testing.

Development Process Costs

From a pure software development perspective, security vulnerabilities identified through security testing can be viewed in the same manner as ”traditional” software bugs that are discovered through standard software testing processes. For the purpose of this discussion, ”traditional” software bugs are those deficiencies identified through non-security-related test functions such as unit- and subsystem-level tests, integration and system-level tests, or stress, performance and load tests.

It is a commonly accepted principle within the software industry that software bugs exposed earlier in the development process are much cheaper to fix than those discovered late in the process. For example, software bugs uncovered by a developer during unit tests generally involve only the developer and require a relatively small amount of effort to diagnose and correct. Conversely, the same software bug identified after the product has been deployed to the customer may involve a large number of personnel and internal processes to diagnose and correct, and therefore may cost significantly more. Examples of these personnel include

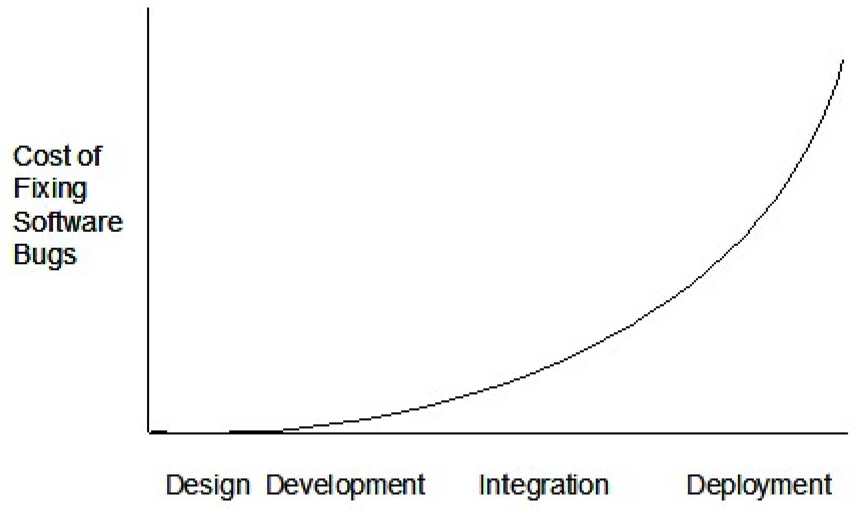
21. Some authors use ”risk-based testing” to denote any kind of testing based on risk analysis. Basing tests on a risk analysis is a sound practice, and we do not mean to denigrate it by adopting a narrower definition.
22. #dsy255-BSI_refs
23. #dsy255-BSI_refs
24. #dsy255-BSI_refs
25. #dsy255-BSI_refs
26. #dsy255-BSI_refs
27. #dsy255-BSI_refs
28. #dsy255-BSI_refs

1. help desk personnel who take trouble calls
2. support engineers who diagnose and confirm the problem
3. developers who identify and implement code fixes
4. QA personnel that perform regression tests
5. support personnel that package and distribute patches
6. mangers that control the process

Support processes required to address and resolve the deficiency include

1. help desk support
2. defect tracking management
3. QA test support
4. patch release management

These additional personnel and processes contribute to the significant expense of correcting software defects after deployment. According to NIST, the relative cost of repairing software defects increases the longer it takes to identify the bug [NIST 02a[49]]. For example, NIST estimates that it can cost thirty times more to fix a coding problem that is discovered after the product has been released than it would have cost if the problem was discovered during unit testing. Other sources, such as IBM, have placed these costs at up to one hundred times more expensive.



Accordingly, the same benefits and cost savings associated with traditional test functions can be realized throughout the software development life cycle with security test functions. Examples of these activities include security tests during the unit, subsystem, and integration test cycles, in addition to security tests during the system test cycle.

As managers are increasingly concerned with controlling and minimizing project costs, security testing throughout the life cycle provides the setting to identify security vulnerabilities early on, when they can be addressed and corrected in a more cost effective manner.

Non-Development Process Business Costs

In addition to the development costs outlined above, it is important to note that security vulnerabilities can have a much greater financial impact on an organization than traditional software bugs. Financial impact can affect both the software development organization and the end customer organization that uses the software.

Security issues attributed to an organization can cause damage to the organization’s reputation or brand, contribute to lost sales or lost customer goodwill, or result in liability and legal issues. As an example,

49. #dsy255-BSI_refs

CardSystem Solutions, a leading provider of payment processing solutions, disclosed in May 2005 that information on 40 million credit cards had been compromised. As a result of the credit card disclosure, Visa and American Express discontinued business with CardSystems, resulting in significant revenue loss for the company. It is likely that the national press coverage will continue to hamper short-term sales prospects for the company. CardSystems has also been named in a class action lawsuit.

Financial losses may also come in the form of reduced stock market capitalization. As reported in the *Journal of Computer Security* [Campbell 03[54]], publicly traded companies that have had information security breaches involving unauthorized access to confidential data may experience a significant negative market reaction. This loss directly and immediately affects company shareholders.

As further evidence, the *2008 CSI/FBI Computer Crime and Security Survey* [Richardson 08[55]] noted that the average financial loss of reporting organizations subjected to theft of proprietary information was $241,000, and those reporting losses due to unauthorized access to information averaged $288,618.

These examples highlight the potential financial impact of security vulnerabilities to the overall business. These vulnerabilities can be addressed by implementing security best practices, including security testing, within the software development life cycle to identify and resolve security issues. Security vulnerabilities that are identified and resolved prior to deployment reduce the overall financial responsibilities and risks to the development and deploying organizations.

Conclusion

The overall goal of security testing is to reduce vulnerabilities within a software system. As described above, these vulnerabilities, at a minimum, will require additional technical troubleshooting and development effort to fix if not exposed before deployment. The cost impact of addressing these vulnerabilities is straightforward to assess. However, in the worst case scenario, security vulnerabilities can adversely impact a company’s reputation or brand, resulting in lost sales or litigation and severely impacting the financial well-being of the business.

Organizations that implement security best practices throughout the software development life cycle understand that addressing issues early can result in cost savings. Security testing is one activity that is used to reduce these vulnerabilities and control potential future costs.

Software Security Testing

At one time, it was widely believed that security bugs in a software system were just like traditional bugs and that traditional software assurance techniques could be equally well applied to secure software development.

However, security-related bugs can differ from traditional bugs in a number of ways:

- Users do not normally try to intelligently search out software bugs. An enterprising user may occasionally derive satisfaction from making software break, but if he or she succeeds it affects only that user. On the other hand, malicious attackers do intelligently search for vulnerabilities. If they succeed, they cause problems for *other* users, who may be adversely affected. Compounding the problem, malicious hackers are known to script successful attacks and distribute them. In other words, a single, hard-to-find vulnerability can cause problems for a large number of users, while a hard-to-find bug causes problems for only a few users.
- Developers can (and do) learn to avoid poor programming practices that can lead to buggy code. In contrast, the list of *insecure* programming practices is long and continues to grow by the year, making it difficult for developers to keep current on the latest exploits. Since most developers are not currently trained in secure programming practices, security analysts carry a greater burden in verifying that secure programming practices are adhered to.

54. #dsy255-BSI_refs
55. #dsy255-BSI_refs

- Security testing is often fundamentally different from traditional testing because it emphasizes what an application should *not* do rather than what it *should* do, as pointed out in [Fink 94[64]], where the authors distinguish between *positive requirements* and *negative requirements*. While security testing may sometimes test conformance to positive requirements such as ”user accounts are disabled after three unsuccessful login attempts” or ”network traffic must be encrypted,” there is a far greater emphasis on negative requirements in security testing. Examples of negative testing include ”outside attackers should not be able to modify the contents of the web page” or ”unauthorized users should not be able to access data.” This shift in emphasis from positive to negative requirements affects the way testing is performed. The standard way to test a positive requirement is to create the conditions in which the requirement is intended to hold true and verify that the requirement is really satisfied by the software. On the other hand, a *negative* requirement may state that something should never occur. To apply the standard testing approach to negative requirements, one would need to create every possible set of conditions, which is infeasible.
- Many security requirements, such as ”an attacker should never be able to take control of the application,” would be regarded as untestable in a traditional software development setting. It is considered a legitimate practice for testers to ask that such requirements be refined or perhaps dropped altogether. But many security requirements can be neither refined nor dropped even if they are untestable. For example, one cannot reliably enumerate the ways in which an attacker might get control of a software system (which would be one way to make it more testable) and obviously one cannot drop the requirement either.

The result is that secure software development is intrinsically harder than traditional software development, and because of this, testing also has an expanded role. Software testing also has other strengths that can be leveraged during secure software development:

- In many software development settings, testing is the only *dynamic* analysis that the software is ever subjected to. A dynamic analysis is one that involves executing the software. Some kinds of problems are easier to detect dynamically than statically, especially problems involving pointers or long-range data and control flow.
- Testing can help confirm that the developers did not overlook some insecure programming practices. Static analysis is useful for this purpose too, but distinguishing true vulnerabilities from unexploitable ones is often easier when dynamic analysis and static analysis are combined.
- A vulnerability is usually taken more seriously if there is a known exploit for it, but developing exploits is the domain of penetration testing (see the BSI module on that topic).
- Testing can be used to help identify and mitigate risks from third-party components, where development artifacts like source code and architecture diagrams are unavailable.
- Testing can be used to provide metrics of software insecurity and help raise the alarm when software is seriously flawed from the security standpoint.
- Every design artifact views the software system at a certain level of abstraction. Clearly, an architecture diagram views the software at an abstract level corresponding to the system architecture, but even the source code is an abstract view of the software. For example, C code does not explicitly show that the strcpy() function can cause a buffer overflow; one has to know in advance that strcpy() is dangerous. Attackers like to find the abstractions used by developers and work their way around them (in other words, find the developers’ implicit assumptions and see what happens when those assumptions are made to be untrue [Hoglund 04[73]], [Whittaker 02[74], Whittaker 03[75]], [Wysopal 07[76]]). No person or group can view a software system at all possible levels of abstraction, but testing can help by perhaps finding (at least some) flaws that are not visible in the design artifacts.

Of course, secure programming is much more than just security testing. In fact, it is often argued that testing is only a small part of the process (see [McGraw 04] or [Howard 06] for example). Nonetheless, testing has a role to play, and given the fact that it is very difficult to find *all* security related problems in a software system, no effective mitigation strategy should be overlooked.

Functional Testing

Functional testing is meant to test a whether a software system behaves is as it should. Usually this means testing the system's adherence to its functional requirements. Since requirements exist at different levels of abstraction throughout the software development process, functional testing also takes place throughout the test process and at different levels of abstraction. For the sake of clarity, it should be noted that many test plans also use ”functional testing” to refer to a specific test phase. For example, it can refer to the testing of executable files and libraries. As is often the case, there is a certain lack of uniformity in the terminology associated with software testing.

Functional testing is a broad topic that the literature on traditional software testing covers in great detail. In this document, we will only discuss this activity in broad strokes, while encouraging test engineers to consult standard references, such as those listed at the end of the Introduction[80], in order to get greater detail. This document also emphasizes the aspects of functional testing that are related to software security.

To understand the nature of functional testing, it is useful to understand the role of functional *requirements* in software development. Requirements generally come from two sources: some are defined up front (e.g., from regulatory compliance or information security policy issues), and others stem from mitigations that are defined as the result of risk analysis. A requirement usually has the following form: ”when a specific thing happens, then the software should respond in a certain way.” For example, a requirement might state that if the user closes an application's GUI window, then the application should shut down. This way of presenting requirements is convenient for the tester, who can simply bring about the ”if” part of the requirement and then confirm that the software behaves as it should. For example, a typical requirement may state that a user's account is disabled after three unsuccessful login attempts or that only certain characters should be permitted in a URL. These requirements can be tested in traditional ways, such as attempting three unsuccessful login attempts and verifying that the account in question was really disabled, or by supplying a URL with illegal characters and making sure they are stripped out before processing.

The tester's life is also simplified by the common practice of ensuring that every requirement can be mapped to a specific software artifact meant to implement that requirement. This helps prevent chaotic code, but it also means that the tester who is probing a specific requirement knows exactly which code artifact to test. Generally, there is a three-way mapping between functional requirements, code artifacts, and functional tests.

Negative requirements create a challenge for functional testing. The mapping of requirements to specific software artifacts is problematic for a requirement such as ”no module may be susceptible to buffer overflows,” since that requirement is not implemented in a specific place. Such requirements are sometimes known as ”negative requirements” because they state things that the software should *not* do, as opposed to what the software *should* do. Often, such requirements can also be expressed as risks, as in ”there is a risk of certain software modules being compromised with buffer overflow attacks.” In this document, negative requirements will be treated separately in the section on risk-based testing.

Although negative requirements often have a high profile in secure software development, one should not overlook the importance of positive requirements in security testing. When risks are identified early enough in the SDLC, there is time to include *mitigations* for those risks (also known as countermeasures). Mitigations are meant to reduce the severity of the identified risks, and they lead to *positive* requirements. For example, the risk of password-cracking attacks can be mitigated by disabling an account after three unsuccessful login attempts, and the risk of SQL insertion attacks from a web interface can be mitigated by using an input validation whitelist that does not contain characters necessary to perform this type of attack. These mitigations also have to be tested, not only to help confirm that they are implemented correctly, but also to help determine how well they actually mitigate the risks they were designed for.

One other issue that deserves special emphasis is that developers might not understand how to implement some security requirements. If a test fails, the developers might also fail to understand what went wrong, and the resulting fix might be a kludge. This can happen with any type of requirement, but security requirements can be a special source of problems, since most developers are not security experts. In one case, it was found

80. #dsy255-BSI_introduction

that a web application was vulnerable to a directory traversal attack, where a URL containing the string ".." can be used to access directories that are supposed to be forbidden to remote clients. After the software was repaired, it was found that developers had simply blacklisted part of the URL used in the original test. Security testers should be aware of common misunderstandings and plan tests for them. When bugs are fixed, it should also be kept in mind that the fix might not be subjected to the same scrutiny as features that were part of the original software design. For example, a problem that should normally be detected in design reviews might slip through the cracks if it appears in a bug fix. Sometimes, software that has been repaired is only retested by running the original test suite again, but that approach works poorly for the kinds of problems just described.

It should be emphasized that functional testing should not provide a false sense of security. Testing cannot demonstrate the absence of problems in software, it can only demonstrate (sometimes) that problems *do* exist [Dijkstra 70[81]]. The problem is that testers can try out only a limited number of test cases, and the software might work correctly for those cases and fail for other cases. Therefore, testing a mitigation is not enough to guarantee that the corresponding risk has really been mitigated, and this is especially important to keep in mind when the risk in question is a severe one.

At the start of this section, we stated that functional testing is meant to probe whether software behaves as it should, but so far we have focused only on requirements-based testing. A number of other techniques are summarized below, and the reader is also referred to the BSI module on white box testing, which covers many of these test techniques in detail.

However, there is one other area of testing still to be covered in this document, namely risk-based testing, which focuses on testing against negative requirements or, in other words, on probing security-related risks. Whether risk-based testing should be regarded as a subset of functional testing is largely a matter of one's taste in terminology, but due to its significant role in secure software development we discuss it separately in Risk-Based Testing.

Some Functional Test Techniques

Ad hoc testing (experience-based testing) and exploratory testing

Derive tests based on tester's skill, intuition, and experience with similar programs. This is also called "exploratory testing." This kind of testing is only effective when done by trained or experienced testers to flesh out special tests not captured in more formal techniques. Ad hoc testing can take advantage of the specialized instincts of security analysts, and it also comes into play when a tester has discovered indirect evidence of a vulnerability and decides to follow up. Penetration testing tends to have an exploratory flavor.

Kaner, C.; Falk, J.; & Nguyen, H. *Testing Computer Software*, 2nd ed., Chapter 1. New York, NY: John Wiley & Sons, 1999.

Bach, J. *Exploratory Testing and the Planning Myth*. http://www.satisfice.com/articles/et_myth.shtml

Marick, B. *A Survey of Exploratory Testing*. http://www.testingcraft.com/exploratory.html

Requirements-based testing

Given a set of requirements, devise tests so that each requirement has an associated test set. Trace test cases back to requirements to ensure that all requirements are covered. In security testing it can also be useful to build test cases around ambiguities in the requirements.

Hamlet, D. & Maybee, J. *The Engineering of Software*, Chapter 7. Boston, MA: Addison-Wesley, 2001.

RTCA, Inc. *DO-178B, Software Considerations in Airborne Systems and Equipment Certification*. Washington, D.C.: RTCA, Inc., 1992. http://www.rtca.org.

81. #dsy255-BSI_refs

Specification-based testing and model-based testing (API testing is a subset)

Given a specification (or even a definition of an interface), test cases can be derived automatically and can even include an oracle. This sometimes requires a specification created in a formal language (which is not often encountered). An alternative form is to create a program model, especially based on interfaces, and derive tests from the interface model. Test cases can also be created by hand based on a specification, but this is much more of an art. In security testing, it can be useful to test situations that are *not* covered in the specifications.

Zhu, H.; Hall, P.; & May, J. ”Software Unit Test Coverage and Adequacy,” Section 2.2. *ACM Computing Surveys 29*, 4 (December 1997): 366-427.

Robinson, H. ”Intelligent Test Automation.” *Software Testing & Quality Engineering* (Sept./Oct. 2000): 24-32. http://www.geocities.com/harry_robinson_testing/Intelligent_Test_Automation.

Equivalence partitioning

Divide the input domain into a collection of subsets, or ”equivalence classes,” which are deemed equivalent according to the specification. Pick representative tests (sometimes only one) from within each class. Can also be done with output, path, and program structure equivalence classes.

Jorgensen, P. C. *Software Testing: A Craftsman’s Approach*, Chapter 6. CRC Press, 1995.

Kaner, C.; Falk, J.; & Nguyen, H. *Testing Computer Software*, 2nd ed., Chapter 7. New York, NY: John Wiley & Sons, 1999.

Hamlet, D. & Maybee, J. *The Engineering of Software*, Chapters 7 and 20. Boston, MA: Addison-Wesley, 2001.

Meyers, G. *The Art of Software Testing*. New York, NY: John Wiley & Sons, 1979.

Boundary value analysis

Choose test cases on or near the boundaries of the input domain of variables, with the rationale that many defects tend to concentrate near the extreme values of inputs. A classic example of boundary-value analysis in security testing is to create long input strings in order to probe potential buffer overflows. More generally, insecure behavior in boundary cases is often unforeseen by developers, who tend to focus on nominal situations instead.

Jorgensen, P. C. *Software Testing: A Craftsman’s Approach*, Chapter 5. CRC Press, 1995.

Kaner, C.; Falk, J.; & Nguyen, H. *Testing Computer Software*, 2nd ed., Chapter 7. New York, NY: John Wiley & Sons, 1999.

Robustness and fault tolerance testing

A variation on boundary value analysis where test cases are chosen outside the domain in order to test program robustness to unexpected and erroneous inputs. Also useful for probing fault tolerance and error handling. Errors can lead to insecure conditions, such as the disclosure of sensitive information in debugging messages or core dumps. Error handlers are also notorious for containing security bugs.

Jorgensen, P. C. *Software Testing: A Craftsman’s Approach*, Chapter 5. CRC Press, 1995.

Kaner, C.; Falk, J.; & Nguyen, H. *Testing Computer Software*, 2nd ed., Chapter 7. New York, NY: John Wiley & Sons, 1999.

Decision table (also called logic-based testing)

Decision tables represent logical relationships between conditions (for example, inputs) and actions (for example, outputs). Derive test cases systematically by considering every possible combination of conditions and actions. Security testers often focus on conditions that are not covered in the requirements or specifications.

Beizer, Boris. *Software Testing Techniques*, Chapter 10. New York, NY: van Nostrand Reinhold, 1990 (ISBN 0-442-20672-0).

Jorgensen, P. C. *Software Testing: A Craftsman's Approach*, Chapter 7. CRC Press, 1995.

State-based testing

Model the program under test as a finite state machine, and then select tests that cover states and transitions using diverse techniques. This is good for transaction processing, reactive, and real-time systems. In security testing, it can often be useful to try to force transitions that do not appear in higher level design artifacts, since vulnerabilities often appear when software enters an unexpected state.

Beizer, Boris. *Software Testing Techniques*, Chapter 11. New York, NY: van Nostrand Reinhold, 1990 (ISBN 0-442-20672-0).

Jorgensen, P. C. *Software Testing: A Craftsman's Approach*, Chapter 4. CRC Press, 1995.

Control-flow testing

Control-flow based coverage criteria aim at covering all statements, classes, or blocks in a program (or some specified combinations). Reduce the program to a directed graph and analyze the graph. Decision/condition coverage is one example. The aim is to detect poor and potentially incorrect program structures. This is often infeasible for all but trivial programs. Coverage analysis is discussed in the BSI module on white box testing.

Beizer, Boris. *Software Testing Techniques*, Chapter 3. New York, NY: van Nostrand Reinhold, 1990 (ISBN 0-442-20672-0).

Jorgensen, P. C. *Software Testing: A Craftsman's Approach*, Chapter 9. CRC Press, 1995.

Hamlet, D. & Maybee, J. *The Engineering of Software*, Chapter 20. Boston, MA: Addison-Wesley, 2001.

Data flow-based testing

Annotate a program control flow graph with information about how variables are defined and used. Use definition-use pairs (often called d/u testing) such that where V is a variable, d is a node where V is defined, and u is a node where V is used and there is a path from d to u. The aim is to detect poor and potentially incorrect program structures. Data flow testing is often used to test interfaces between subsystems. Data-flow analysis is discussed in the BSI module on white box testing.

Beizer, Boris. *Software Testing Techniques*, Chapter 5. New York, NY: van Nostrand Reinhold, 1990 (ISBN 0-442-20672-0).

Hamlet, D. & Maybee, J. *The Engineering of Software*, Chapter 17. Boston, MA: Addison-Wesley, 2001.

Usage-based and use-case based testing

Base tests on use of the product in real operation by creating an operational profile or creating a set of use cases. It is sometimes possible to infer future reliability from test results (given a statistically correct operational profile). Do this by assigning inputs to a probability distribution according to their occurrence in actual operation.

Jorgensen, P. C. *Software Testing: A Craftsman's Approach*, Chapter 14. CRC Press, 1995.

Lyu, M. *Handbook of Software Reliability Engineering*, Chapter 5. McGraw-Hill/IEEE, 1996. http://www.cse.cuhk.edu.hk/~lyu/book/reliability/.

Pfleeger, S. L. *Software Engineering: Theory and Practice*, Chapter 8. Upper Saddle River, NJ: Prentice Hall, 1998 (ISBN 013624842X).

Cockburn, A. Usecases.org. http://alistair.cockburn.us/Use+cases

Wiegers, K. *Software Requirements*. Redmond, WA: Microsoft Press, 1999.

Code-based testing (also called white box testing)

Use the control structure, the data flow structure, decision control, and modularity to design tests to cover the code. Use coverage analysis (e.g., white box) to assess test completeness and goodness. This technique is a superset of control flow testing and data flow testing. White box testing is covered in a separate module of the BSI portal.

Beizer, Boris. *Software Testing Techniques*, Chapter 3. New York, NY: van Nostrand Reinhold, 1990 (ISBN 0-442-20672-0).

Jorgensen, P. C. *Software Testing: A Craftsman's Approach,* Chapter 4. CRC Press, 1995.

Hamlet, D. & Maybee, J. *The Engineering of Software*, Chapter 20. Boston, MA: Addison-Wesley, 2001.

Marick, Brian. *The Craft of Software Testing: Subsystems Testing Including Object-Based and Object-Oriented Testing*. Upper Saddle River, NJ: Prentice Hall PTR, 1994.

Fault-based testing

Intentionally introduce faults during testing to probe program robustness and reliability. Determining which kind of faults to introduce and how to observe their effects is a challenge. Experience with this method is necessary for it to be useful. Code-based fault injection is discussed in the BSI module on white box testing.

Voas, Jeffrey M. & McGraw, Gary. *Software Fault Injection: Inoculating Programs Against Errors*, 47-48. New York, NY: John Wiley & Sons, 1998.

Protocol conformance testing

Use a program's communication protocol as a direct basis for testing the program. This is useful when a program is supposed to accept a protocol. In combination with boundary-value testing and equivalence-based testing, this method is useful for web-based programs and other Internet-based code. Protocol-based testing is especially important for security testing in web-based applications, since the easiest way for remote attackers to access such applications is through web protocols (or their buggy implementations, as the case may be). Protocol-based testing is discussed in the BSI portal on black box tools.

Sun, Xiao; Feng, Chao; Shen, Yinan; & Lombardi, Fabrizio. *Protocol Conformance Testing Using Unique Input/Output Sequences*. Hackensack, NJ: World Scientific Publishing Co., 1997.

Load and performance testing

Testing specifically aimed at verifying that the subsystem meets specified performance requirements (e.g., capacity and response time). Load and stress testing exercise a system to the maximum design load and beyond it. Stressful conditions can expose vulnerabilities that are otherwise hard to see, and vulnerabilities can also be caused by the mechanisms that software uses to try to deal with extreme environments. Developers are often focused on graceful degradation when they create these mechanisms, and they overlook security.

Perry, W. *Effective Methods for Software Testing*, Chapter 17, New York, NY: John Wiley & Sons, 1995.

Pfleeger, S. *Software Engineering Theory and Practice,* Chapter 8, Upper Saddle River, NJ: Prentice Hall PTR, 1998.

Security testing

The use of a variety of testing techniques specifically to probe security. There are two major aspects of security testing: testing security functionality to ensure that it works and testing the subsystem in light of malicious attack. Security testing is motivated by probing undocumented assumptions and areas of particular complexity to determine how a program can be broken.

Howard, Michael & LeBlanc, David C. *Writing Secure Code,* 2nd ed. Redmond, WA: Microsoft Press, 2002, ISBN 0735617228.

Howard, Michael & Lipner, Steve. *The Security Development Lifecycle*. Redmond, WA: Microsoft Press, 2006, ISBN 0735622142.

Viega, John & McGraw, Gary. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley Professional, 2001 (ISBN 020172152X).

Wysopal, Chris; Nelson, Lucas; Zovi, Dino Dai; & Dustin, Elfriede. *The Art of Software Security Testing*. Upper Saddle River, NJ: Symantec Press, 2007, ISBN 0321304861.

Run-Time Verification

Run-time verification seeks to validate that an application conforms to its security requirements and specifications by dynamically observing the application's behavior in a test environment. Requirements such as "all authentication credentials must be encrypted while in transit" can thus be dynamically verified through observation.

Howard, Michael & Lipner, Steve. *The Security Development Lifecycle*. Redmond, WA: Microsoft Press, 2006, ISBN 0735622142.

Risk-Based Testing

Recall that in security testing, there is an increased emphasis on *negative* requirements, which state what a software system should not do. Tests can be developed in a number of ways for negative requirements. The tests should be derived from a risk analysis, which should encompass not only the high-level risks identified during the design process but also low-level risks derived from the software itself.

When negative requirements are tested, security testers typically look for common mistakes and test suspected weaknesses in the application. The emphasis is often on finding vulnerabilities, often by executing abuse and misuse tests that attempt to exploit the weaknesses in the application. In addition to demonstrating the presence of vulnerabilities, security tests can also assist in uncovering symptoms that suggest vulnerabilities might exist.

It was stated earlier that requirements can be expected to contain *mitigations* for many risks. Mitigations generally result in positive requirements, but the fact that some risk has a mitigation does not imply that it should be ignored during risk-based testing. Even if a mitigation is correctly implemented, there is still a need to ask whether it really does mitigate the risk it is intended for. Each mitigation generates a positive requirement—the correct implementation of the mitigation strategy—but it also generates a negative requirement stating that the mitigation must not be circumventable. To put it another way, the mitigation might not be sufficient for avoiding the underlying risk, and this possibility constitutes a risk in and of itself.

Unfortunately, the process of deriving tests from risks is somewhat of an art, and depends a great deal on the skills and security knowledge of the test engineer. There are many automated tools that can be helpful aids during risk-based testing [Black Box Testing[180]], but these tools can perform only simple tasks, while the hard tasks are still the responsibility of the test engineer. The process of test creation from negative requirements is discussed in greater detail below.

Defining Tests for Negative Requirements

As a basis for defining test conditions, past experience comes into play in two ways. First, a mature test organization will have a set of *test templates* that outline the test techniques to be used for testing against specific risks and requirements in specific types of software modules. Test templates are usually created during testing projects, and they accumulate over time to record the organization's past experience. This document does not provide test templates, but the attack scenarios of [Hoglund 04[184]], [Whittaker 02[185]], [Whittaker 03[186]], and [Wysopal 07[187]] can be used for this purpose.

180. http://buildsecurityin.us-cert.gov/bsi/articles/tools/black-box.html (Black Box Testing)
184. #dsy255-BSI_refs
185. #dsy255-BSI_refs
186. #dsy255-BSI_refs
187. #dsy255-BSI_refs

Another way to derive test scenarios from past experience is to use incident reports. Incident reports can simply be bug reports, but in the context of security testing they can also be forensic descriptions of successful hacking activity. Furthermore, vulnerability reports are often followed by proofs of concept to demonstrate how the reported vulnerability is actually exploitable. Sometimes these proofs of concept are actual exploits, and sometimes they simply show that a vulnerability is *likely* to be exploitable. For example, if a buffer overflow can be made to cause a crash, then it can usually also be exploited. Sometimes it is enough simply to find evidence of vulnerabilities as opposed to actual exploits, so these proofs of concept can be used as the basis of test scenarios. When devising risk-based tests, it can be useful to consult IT security personnel, since their jobs involve keeping up to date on vulnerabilities, incident reports, and security threats.

Finally, threat modeling can be leveraged to help create risk-based tests. For example, if inexperienced hackers (e.g., script kiddies) are expected to be a major threat, then it might be appropriate to probe software security with automated tools; hackers often use the same tools. Some of these tools are described in the BSI module on black box testing tools. Threat modeling should be carried out in any event as part of a secure software development process, and it is described in its own BSI module.

Below, we discuss some thought processes that may be helpful in creating new tests for negative requirements.

Understanding a Software Component and Its Environment

When testing against negative requirements, one of the test engineer's first tasks is to understand the software and its environment. It is important to understand the software itself, but it is also important to understand the environment, and in fact this issue may deserve even more emphasis because not all interactions with the environment are obvious. A software module may interact with the user, the file system, and the system memory in fairly obvious ways, but behind the scenes many more interactions may be taking place without the user's knowledge. This understanding can be aided by a controlled environment like the one described above, where software behavior can be observed in detail.

Two types of environmental interactions have to be considered, namely, corruption of the software module by the environment and corruption of the environment by the software module. Regarding the environment as a potential threat is part of defense in depth, even when considering parts of the environment that have their own protection mechanisms. As for corruption of the environment, recall that there may not be enough time to test every component of the software system, and that the components not currently under test are part of the environment. If the component under test can corrupt the system—perhaps in cooperation with other components that happen to be in an unexpected state—this constitutes a direct threat to the system just as if the component itself were vulnerable.

Some environmental interactions to be considered are

- user interactions
- interactions with the file system
- interactions with memory
- interactions with the operating system via system calls
- interactions with other components of the same software system
- interactions with dynamically linked libraries
- interaction with other software via APIs
- interaction with other software via interprocess communication
- interactions with global data such as environment variables and the registry
- interactions with the network
- interactions with physical devices
- dependencies on the initial environment

The biggest challenge is to avoid overlooking an interaction altogether; that is, remembering to even ask whether the software can or should trust the entity that it is interacting with. Still, it is appropriate to prioritize these interactions according to how realistic it is to view them as part of an attack. For example, the operating system is normally in charge of protecting real memory, and if it fails to do so this is regarded as a security bug, so the main goal is to check for ways that the attacker might corrupt real memory rather than assuming it to be corrupted. Similarly, the tester might well assign low priority to attacks where the attacker would have to break a properly validated encryption scheme.

Understanding the Assumptions of the Developers

Attackers attack the assumptions of developers. That is, developers have a certain mental view of the software, and that mental view does not cover every possible thing that the software can do. This state of affairs is inevitable because software is too complex for a human to carry a complete, detailed mental picture of it. Nonetheless, the tester's job is to find the developer's assumptions, violate those assumptions, and thereby try to uncover vulnerabilities. Often, developers' assumptions come in the form of *abstractions* that can be anything from a design diagram to a datatype to a stereotypical view of how the software will be used. An abstraction can predict and explain certain software behaviors, but by nature there are other things that it does *not* predict or explain. Part of the tester's job is to *break* the abstraction by forcing behaviors that the abstraction does not cover.

Any mental picture is an abstraction; by nature it hides some details in order to provide a coherent big picture. The designers and developers also had certain abstract views of the software in mind, as did previous testers, even though those abstractions might not always have been selected with much conscious thought. For example, an architecture diagram shows the software at one level of abstraction, the high-level source code shows it at another, and the machine code at still another. Interactions between the software and the environment are represented as abstractions too.

A program written in a high-level language is an abstract representation of the actual software behavior, but if there is a compiler bug it may not reflect the software's true behavior. In C/C++, the notorious string buffer is an abstraction that fails to reveal how the characters in a string can affect the program counter. A high-level architecture diagram is a different kind of abstraction that does not show how an attacker might interfere with communication between modules. Therefore, developers whose abstract view of a system is based on the architecture diagram often fail to foresee such attacks.

This relates to the process of test creation in another way: the process of devising software tests is driven by the tester's *own* mental picture of how the software system works. For example, [Whittaker 02[210]] suggests (in the context of ordinary testing) that testers adopt an abstraction where the software has four classes of "users," namely, the file system, the kernel, human users, and other applications. In this abstraction, program inputs and outputs are mediated by the kernel. Within this view, a tester's mental picture of program behavior depends on whether the software is viewed as high-level code, object code, machine code, or perhaps even microcode. At a completely different level of abstraction we might say that the software does not receive input at all but rather supplies instructions to a microprocessor, which in turn receives input from other hardware.

In security testing, there is no "correct" level of abstraction because any manageable abstraction hides something, and whatever is hidden might turn out to be exploitable by an attacker. Instead of trying to find one mental abstraction that is somehow better than the others, the goal is simply to avoid being dogmatic about this issue and instead try to find behaviors and interactions that were previously overlooked.

Building a Fault Model

A *fault model* consists of hypotheses about what might go wrong with a piece of software [Leveson 95[213]]. In other words, the tester must ask what kinds of vulnerabilities can exist in a system of this type and what kinds of problems are likely to have been overlooked by the developers.

Often the most important classes of vulnerabilities to consider may be the most common ones, that is, the vulnerabilities targeted by security scanners, reported in public forums, and so on. These include buffer

210. #dsy255-BSI_refs
213. #dsy255-BSI_refs

overflow vulnerabilities, format string vulnerabilities, cross-site scripting vulnerabilities, and so on. These kinds of vulnerabilities are undesirable in almost any setting. IT security personnel can also be a useful resource for keeping track of current vulnerabilities.

Application-specific security requirements lead to a second class of potential vulnerabilities, where an attacker has the ability to circumvent or otherwise undermine application-specific security. The fact that these vulnerabilities are not common to a large class of applications makes it more difficult to hypothesize about specific faults, since there is less experience to draw on both in the public domain and (most likely) in the tester's own body of past experience. Of course, it is extremely valuable if the tester can find past work on similar systems, and this potential avenue should not be ignored, but it is more predictable to start with the risk analysis. Significant portions of the risk analysis might already be based on examination of the software—the analysis may document potential bugs—so it may already constitute a significant part of the fault model.

Finally, many traditional software bugs can also have security implications. Buggy behavior is almost by definition unforeseen behavior, and as such presents an attacker with the opportunity for a potential exploit. Indeed, many well-known vulnerabilities, such as buffer overflow, format-string, and double-free vulnerabilities, could cause a software crash if they were triggered accidentally rather than being exploited. Crashing software can also expose confidential information in the form of diagnostics or data dumps. Even if the software does not crash as the result of a bug, its internal state can become corrupted and lead to unexpected behavior at a later time. Finally, error handlers themselves are a frequent target of malicious attacks. Attackers probing a new application often start by trying to crash it themselves. For these reasons, traditional software faults have to be considered during security testing as well.

One important difference between security testing and other testing activities is that the security tester is emulating an intelligent attacker. This has several implications. Most importantly, an adversary might do things that no ordinary user would do, such as entering a thousand-character surname or repeatedly trying to corrupt a temporary file. Security testers must consider actions that are far outside the range of normal activity and might not even be regarded as legitimate tests under other circumstances. Secondly, an attacker will go straight for the program's weak spots, so the tester must think like the attacker and find the weak spots first. Finding ninety percent of an application's vulnerabilities does not make the application less vulnerable; it only reduces the cost of future fixes and increases the odds of finding the remaining problems before attackers do.

Needless to say, testers must also be aware of other standard attacks such as buffer overflows and directory-traversal attacks. Even though commercial software is increasingly immune to such attacks, this immunity stems largely from increased caution on the part of development organizations. The security tester shares responsibility for sustaining this state of affairs.

Other vulnerabilities may be no less obvious even though they are specific to a given application. For example, a developer may simply *trust* input from a given source, without having seriously considered the possibility that the input might be corrupted by an attacker. Applications might also *write* information that is trusted elsewhere without considering whether the attacker can influence what gets written.

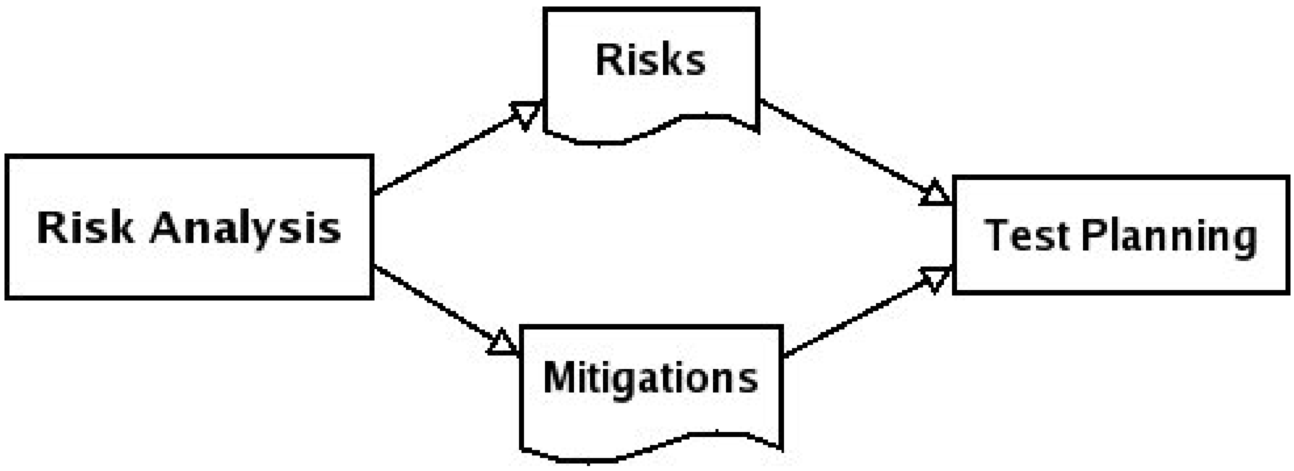
Security Testing in the Software Life Cycle

Activities related to testing take place throughout the software life cycle. Preparatory activities, especially test planning, take place even before there are any artifacts to test.

At a high level, it is useful to think of software testing as being embodied by the *test plan,* since it describes the entire test process. Like risk analysis [Risk Management[217]], test planning is *holistic* in that it takes place throughout the software development process, and *fractal* in that similar activities take place at different levels of abstraction.

Figure 1. A simplified diagram showing the relationship between risk analysis and test planning

217. http://buildsecurityin.us-cert.gov/bsi/articles/best-practices/risk.html (Risk Management)



Test planning—and hence testing—is intrinsically related to risk analysis. Risk analysis also goes on throughout the development process, and it leads not only to the identification of risks but also to the definition of *mitigations* for those risks. It is often a good idea to devise tests that probe specific risks identified during risk analysis; this leads to risk-based testing. On the other hand, it is also necessary to devise tests for mitigations. These are usually functional tests, and they help determine whether the mitigations have been implemented correctly or implemented at all. Since risk analysis is an ongoing and fractal process throughout software development, new information is always becoming available for the test plan, and test planning becomes an ongoing process as well.

The Start of the Software Life cycle

Usually, the first stage in the software life cycle is when an organization determines the need for a software product. Sometimes [Grance 04[224]] this is called the initiation phase of the software life cycle. In a sense, this phase defines what the software product is going to be.

Preparations for security testing can start even before the planned software system has definite requirements and before a risk analysis has begun. For example, past experience with similar systems can provide a wealth of information about how attackers have typically tried to subvert similar systems in the past.

More generally, the initiation phase makes it possible to start a preliminary risk analysis, asking what environment the software will be subjected to, what its security needs are, and what impact a breach of security might have. This information provides an early focus for the process of test planning. If risk analysis starts early, it will also be possible to take a security-oriented approach when defining requirements.

Security Testing and the Requirements and Design Phases

The process of software development starts by gathering requirements and developing use cases. In this phase, *test planning* focuses on outlining how each requirement will be tested. Some requirements may appear to be untestable, and if test planning is already underway, then those requirements can be identified and possibly revised to make them testable.

It is important to remember that test planning involves more than just laying out a series of tests. For example, the test plan must also encompass test management and test automation, map out the various phases of the test process including entry and exit criteria, and provide an estimate of the resources needed for each activity (these aspects of test planning will be discussed in greater detail below). A good test process requires many preliminary activities before testing can begin, and some of those activities can take time and/or be subject to delays. Mapping out the elements of the test plan should begin in the requirements phase of the development life cycle in order to avoid surprises later on.

It is also useful for a more detailed risk analysis to begin during the requirements phase. Testing is driven by both risks and requirements, and risks are especially important to consider in security testing. While traditional non-security related risks are linked to what can go wrong if a requirement is not satisfied,

224. #dsy255-BSI_refs

security analysis often uncovers severe security risks that were not anticipated in the requirements process. In fact, a security risk analysis is an integral part of secure software development, and it should help drive requirements derivation and system design as well as in security testing.

The risks identified during this phase may lead to additional requirements that call for features to mitigate those risks. Mitigations are solutions that are developed to address a particular security risk. The software development process can be expected to go more smoothly if mitigations are defined early in the life cycle, when they can be more easily implemented.

The testing process is based on developing test cases for mitigations as well as risks and requirements. If mitigations are planned for a particular risk, then security testing focuses on those mitigations as well as the underlying risk itself. If there is time pressure, it is often a legitimate strategy to spend less time testing against a risk that has a mitigation, on the assumption that a mitigated risk is less severe. For example, suppose the application being developed is a web server, and it is determined that there is a risk of injection attacks. Injection attacks involve the insertion of special characters into URLs, so the usual mitigation for this risk is to preprocess each URL to ensure that it only contains characters from a certain ’legal’ character set. That requirement can be tested by verifying that the offending characters really are being rejected. Ultimately, the software design may call for a specific module that performs the necessary preprocessing work, and this makes the tester’s task simpler because the necessary tests might only have to be conducted on that particular module.

The example above illustrates how mitigations can turn a negative requirement into a positive requirement. The negative requirement is ”the web server should not be susceptible to injection attacks” and the positive one is ”only legal characters should be permitted in a URL,” where the definition of ’legal’ is also part of the requirement. It is often the case that positive requirements are easier to test than negative ones. Of course, mitigations should not lead to a false sense of security; after all, the system might be susceptible to injection from other vectors than URLs, or else a software flaw might make it possible to bypass the preprocessing module. In other words, there might still be negative requirements that have to be tested. However, the presence of the mitigation lets testers spend less time trying out obvious injection attacks—only a few may be needed to confirm that the mitigation was implemented correctly—and therefore the testers can spend more time looking for more subtle injection vulnerabilities. The presence of the mitigation changes what has to be tested (and in most cases it also creates greater confidence in the final product).

Security Testing in the Test/Coding Phase

Functional security testing generally begins as soon as there is software available to test. A test plan should therefore be in place at the start of the coding phase and the necessary infrastructure and personnel should be allocated before testing starts.

Software is tested at many levels in a typical development process, though the actual test activities may vary from project to project and from organization to organization. For example, the first test stage often focuses on individual functions, methods, or classes and is known as *unit testing*. Subsequently, there might be a stage for testing modules that represent individual libraries or individual executables, and these might be tested for their adherence to the requirements as well as for correct integration with one another. The next stage might be to test small subsystems containing several executable modules and/or libraries, again checking adherence to requirements and integration. Eventually the entire system is ready to be tested as a whole, and it too may be tested against the requirements as well as being tested for proper integration. The complete system may also go through a number of subsequent test stages, including tests for usability, tests in the operational environment, and stress testing. The above is not meant to be a description of all test processes, much less a *pre*scription for how to test; it is simply meant to give the reader a feeling for what a test process might consist of.

This document does not attempt to catalog every possible testing activity. Instead, it will discuss several broader activities that are common to most test processes, some of which are repeated at different times for components at different levels of complexity. We will discuss the role of security testing in each of these activities. The activities discussed below are

- unit testing, where individual classes, methods, functions, or other relatively small components are tested
- testing libraries and executable files
- functional testing, where software is tested for adherence to requirements
- integration testing, whose goal is to test whether software components work together as they should
- system testing, where the entire system is under test

Unit Testing

Unit testing is usually the first stage of testing that a software artifact goes through. There is no fixed definition of what a ”unit” is for the purposes of unit testing, but usually the term connotes individual functions, methods, classes, or stubs. Since unit testing generally involves software artifacts that cannot be executed by themselves, it usually requires test drivers. Unit testing is by nature functional testing, since there is nothing to integrate yet. It is listed separately here because it differs in other fundamental ways from later test activities.

During unit testing, the emphasis is usually on positive requirements. Positive requirements state what software should do as opposed to saying what it should not do. The unit under test is usually there to implement one or more requirements, and the goal is to confirm that it really does implement those requirements.

Generally, unit testing is integrated into the development setting. The responsibility for unit testing often falls on the shoulders of those who are most capable of writing the test drivers, namely, the developers themselves. In some cases there are also dedicated testers who interact with individual developers and test the artifacts of those developers.

This integration of unit testing and software development is not only motivated by the need for test drivers, it also makes the test process more efficient. Later stages of testing might make use of limited resources such as specialized tools and a customized test environment. It is not efficient to use these resources for finding bugs that are immediately obvious once the software is executed, especially if those bugs prevent further testing. This says nothing of the time that is lost by sending buggy software back and forth between the test and development environments countless times in order to fix simple and obvious bugs.

In spite of being the responsibility of the developers, a discussion of which security principals must be included in unit tests should be included in the test plan. This helps ensure that developers know what they must test, that testers know what they should expect from the developers, and that the developers have the resources *they* need to do that part of their job.

When planning unit tests for security, care should be taken not to underestimate the possible security threats to components deep inside an application. It must be kept in mind that the *attacker's* view of the software environment may be quite different from that of the ordinary user. In other words, the attacker might be able to get at the software in ways that an ordinary user might not. For example, if a component reads data from a file, it might be appropriate for that component to validate that data on the assumption that an attacker might have corrupted it. If a component makes assumptions about inputs provided by a user—even when those inputs are being checked elsewhere—it may be appropriate to validate the assumptions in the component itself, because these assumptions are best understood by the component’s developers. Checking such assumptions locally also simplifies later maintenance in case those assumptions change.

When evaluating potential threats to a software component, one should also keep in mind that many attacks have two stages: one stage where a remote attacker gains initial access to the machine, and one stage where the attacker gains *control* of the machine after gaining access. For example, an attacker who subverts a web server will initially have only the privileges of that web server, but the attacker can now use a number of *local* attack strategies such as corrupting system resources and running privileged programs in corrupt environments. Therefore, defense in depth demands that local threats be addressed on systems that are assumed not to have malicious users, and even on machines that supposedly have no human users at all. Here again, it is appropriate to ask what assumptions a component makes about its environment and whether those assumptions are being checked. Attack trees [Schneier 00b[250]] are a common method used to identify and model security threats that require multiple stages to execute.

Testing Libraries and Executable Files

In many development projects, unit testing is closely followed by a test effort that focuses on libraries and executable files.

Usually test engineers, rather than software developers, perform testing at this level. A greater level of testing expertise is needed, and this is particularly true of security testing because the testers need to be up to date on the latest vulnerabilities and exploits. More generally, security testing is a specialized talent, and it may be too expensive to hire full-time software developers that have this talent in addition to being skilled at development. The test environment can also be complex, encompassing databases, stubs for components that are not yet written, and complex test drivers used to set up and tear down individual test cases. In security testing, there may also be a need for specialized technology that crafts customized network traffic, simulates fault and stress conditions, allows observation of anomalous program behavior, and so on. Aside from the fact that this technology would be expensive to replicate throughout the development environment, building and using it may also require the specialized expertise of a test engineer.

It is useful during functional testing to carry out code coverage analysis using a code coverage tool. This helps in isolating program parts not executed by functional testing. These program parts may include functions, statements, branches, conditions, etc. Such an analysis helps to

- focus and guide testing procedures
- assess thoroughness of testing
- identify code not executed by tests
- check adequacy of regression test suites
- keep in sync with code changes

Coverage analysis can be especially important in security testing. Since a determined attacker will probe the software system thoroughly, security testers must do so as well. Error handling routines are notoriously difficult to cover during testing, and they are also notorious for introducing vulnerabilities. Good coding practices can help reduce the risks posed by error handlers, but it may still be useful to have testware that simulates error conditions during testing in order to exercise error handlers in a dynamic environment.

Libraries also need special attention in security testing. Components found in a library might eventually be reused in ways that are not evident in the current system design. Libraries should be tested with this in mind: just because a library function is protected by other components in the current design does not mean that it will always be protected in the future. For example, a buffer overflow in a particular library function may seem to pose little risk because attackers cannot control any of the data processed by that function, but in the future the function might be reused in a way that makes it accessible to outside attackers.

Furthermore, libraries may be reused in future software development projects, even if this was not planned during the design of the current system. This creates additional problems. First, the persons who developed the library code might not be available later, and the code may not be well understood anymore. This will make security testing harder when the library is reused, so initial testing should be thorough. Secondly, vulnerabilities in the library will have a greater negative impact if the library is reused in many systems. Finally, if the library is used widely, malicious hackers might become familiar with its vulnerabilities and have exploits already at hand. This makes it especially important to audit and test library functions early on. Perhaps the most notorious example of a vulnerable library function is the strcpy()function in the standard C library, which is susceptible to buffer overflows. For many years, calls to strcpy() and similarly vulnerable functions were one of the chief causes of software vulnerabilities in deployed software. By the time strcpy()'s vulnerability was discovered, the function was in such widespread use that removing it from the standard C library would have been infeasible. The story of strcpy() contains a moral about the value of catching vulnerabilities as soon as possible and the expense of trying to address vulnerabilities late in the

250. #dsy255-BSI_refs

life cycle. (On the other hand, it is hard to imagine how the authors of strcpy() could have foreseen these problems in the 1960s, so the there is also a moral about the limitations of secure software engineering in the face of unknown classes of vulnerabilities.)

Integration Testing

Integration testing focuses on a collection of subsystems, which may contain many executable components. There are numerous software bugs that appear only because of the way components interact, and this is true for security bugs as well as traditional ones.

Integration errors are often the result of one subsystem making unjustified assumptions about other subsystems. A simple example of an integration error occurs when library functions are called with arguments that have the wrong data type. In C, for example, there need not be a compiler warning if an integer value is passed where an unsigned integer value is expected, but doing so can change a negative number to a large positive number. One way that this can lead to a vulnerability is if the variable in question is used as a buffer length; it can circumvent whatever bounds checking was done by the calling function. An integration error can also occur if the caller and the callee both assume that the other function was responsible for bounds checking and neither one actually does the check.

In fact, the failure to properly check input values is one of the most frequent sources of software vulnerabilities. In turn, integration errors are one of the most frequent sources of unchecked input values, because each component might assume that the inputs are being checked elsewhere. (It was mentioned earlier that components should validate their own data, but in many systems this is an ideal that has to be sacrificed for reasons of efficiency.) During security testing, it is especially important to determine what data can and cannot be influenced by a potential attacker.

Error handlers were already mentioned above in the section on functional testing, but they can also lead to integration errors because of unusual control and data-flow patterns during error handling. Error handling is one more form of component interaction, and it must be addressed during integration testing.

System Testing

In the late stages of a software development process, the entire system is available for testing. This testing stage can (and should) involve integration and functional tests, but it is treated separately here because the complete system is the artifact that will actually be attacked. Furthermore, certain activities relevant to software security, such as stress testing, are often carried out at the system level. Penetration testing is also carried out at the system level, and when a vulnerability is found in this way there is tangible proof that the vulnerability is real; a vulnerability that can be exploited during system testing will be exploitable by attackers. Under resource constraints, these are the most important vulnerabilities to fix, and they are also the ones that will be taken most seriously by developers.

It was stated above that the complete system is the artifact that will be attacked. This is true even though an attacker can usually execute individual components once he or she gains access to a local machine. In order to gain that access it is often necessary to first subvert an outward-facing software system. The most common example is a system that provides some network service and is therefore accessible to the world at large, but there are also other cases where an attacker is forced to deal with an entire software system. For example, an attacker in one protection domain may be attacking another, or an attacker who controls one layer of an operating system might be attacking another layer. While defense in depth demands that individual executable programs be secure, it also demands rigorous security at the system level to help prevent attackers from getting a toehold in the first place.

Often, penetration testing is associated with system-level testing. Penetration testing makes the most sense here, because any vulnerabilities it uncovers will be real vulnerabilities. In contrast, earlier test stages take place in an artificial environment that might not represent the true environment closely enough. Furthermore, some system components might be represented by stubs in earlier test stages. The case study in the Case Study[272] section describes a system where poor version control caused a fault to remain hidden until system

272. #dsy255-BSI_case-study

testing. The moral of these examples is the importance of testing the actual artifact that will be deployed, especially in activities like penetration testing, where the goal is to uncover real vulnerabilities.

Stress testing is also relevant to security because software performs differently when under stress. For example, when one component is disabled due to insufficient resources, other components may compensate in insecure ways. An executable that crashes altogether may leave sensitive information in places that are accessible to attackers. Attackers might be able to spoof subsystems that are slow or disabled, and race conditions might become easier to exploit. Stress testing may also exercise error handlers, whose importance was already emphasized above. Security testers should also look for unusual behavior during stress testing that might signal the presence of unsuspected vulnerabilities. Stress testing is a time when components are likely to have to deal with unexpected situations, and many vulnerabilities come about because of conditions that developers were not expecting.

The importance of functional and integration testing at the system level should not be overlooked. During earlier test phases, some components are likely to have been replaced by stubs, and system testing is usually the first time that the system actually does the same things it will do after deployment.

The Operational Phase

The operational phase of the software life cycle begins when the software is deployed. Often, the software is no longer in the hands of the developing organization or the original testers.

Traditionally, beta testing is associated with the early operational phase, and beta testing has its counterpart in the security arena, since white-hat hackers may examine the software in search of vulnerabilities.

However, beta testing is not the last word because the software system may *become* vulnerable after deployment. This can be caused by configuration errors or unforeseen factors in the operational environment. Vulnerabilities can also creep into an existing system when individual components are updated. Assumptions that were true about those components during development may no longer be true after new versions are deployed. It may also be that only some components are updated while others are not, creating a mélange of software versions whose exact composition cannot be foreseen in any development setting. This makes the vulnerabilities of the composite system unforeseeable as well. Some subsystems might also be customized on site, making it harder still to maintain a big picture of the software system and its potential vulnerabilities

The risk profile of the system might also change over time, and this creates a need for continuing security audits on systems deployed in the field. This can happen when the system is used in ways that were not foreseen during development, or when the relative importance of different assets grows or diminishes. For example, an organization might unknowingly protect critical information with encryption methods that are no longer considered secure.

The risk profile can also change when new vulnerabilities are uncovered in existing software versions. In fact, there may be entirely new *classes* of vulnerabilities that were not foreseen during development. For example, a *format string vulnerability* exists when an attacker can control the first argument of a C/C++ print statement (these vulnerabilities are discussed elsewhere in the BSI portal). Before they were recognized as vulnerabilities, it is difficult to imagine how any development effort, no matter how meticulous, could have systematically avoided them. The danger of a simple statement like print(working_directory) was simply not recognized. In general, malicious attackers know about vulnerabilities soon after they are discovered. A responsible development organization will release a patch, but software users might not *apply* the patch. This creates a drastic shift in the system’s risk profile.

A related issue is that *encryption techniques* used by a software system can become obsolete, either because increasing computational power makes it possible to crack encryption keys by brute force or because researchers have discovered ways of breaking encryption schemes previously thought to be secure.

These issues create the need for security audits in deployed systems. Ideally, the audits should be performed by security professionals, and many test activities, especially those associated with system testing, can be useful here too.

Unfortunately, security audits often consist of checklists and scans by automated tools. This makes it necessary to leave the discussion of security testing in the software life cycle on a somewhat negative note. Too often, organizations place too much faith in weak, automated testing tools as the sole means of conducting security audits. Such testing tools usually run a series of canned tests simulating known attacks, although to be fair they also check software versions and run other checks that do not directly involve penetration attempts. These tools have their place, but they should not be the sole means of maintaining security in a deployed system.

Security Testing Activities

This section discusses several activities associated with software testing, emphasizing the role of security. It is not intended as a specification for a test process, but it highlights several parts of the process that are especially relevant to security testing.

Risk Analysis

The main ingredient of a secure software development process is risk analysis. Risk analysis serves two main purposes in testing: it forms the basis for risk-based testing, which was discussed in the Risk-Based Testing[286] section, and it also forms the basis for test prioritization.

The need for test prioritization arises because there is rarely enough time to test as thoroughly as the tester would like, so tests have to be prioritized with the expectation that tests with lower priority may not be executed at all. Risk analysis can be used to rank test activities, giving priority to tests that address the more important risks.

The technical risks identified in the risk analysis should identify threats and vulnerabilities to the system to guide testing effort. The risks identified should be used to

- develop an overall test strategy, which includes defining scope, selecting applicable testing techniques, defining acceptable test coverage metrics, defining the test environment, etc.
- develop particular tests based on threats, vulnerabilities, and assumptions uncovered by the analysis. For example, tests could be developed to validate specific design assumptions or to validate controls (or safeguards) put in place to mitigate certain risks.
- increase test coverage and test focus in risky areas identified by the analysis, specifically vulnerable portions of the software. For example, a specific component or functionality may be more exposed to untrusted inputs, or the component may be highly complex, warranting extra attention.
- select test data inputs based on threats and usage profiling created by risk analysis. For example, analysis could reveal that the system may be vulnerable to a privilege escalation problem.

Risk analysis is discussed elsewhere in the BSI portal, but it is mentioned here because it is an important prerequisite for the other activities discussed in this section.

Creating a Test Plan

The main purpose of a test plan is to organize the security testing process. It outlines which components of the software system are to be tested and what test procedure is to be used on each one. A test plan contains more than a simple list of tests that are to be carried out. In general, the test plan should incorporate both a high-level outline of which artifacts are to be tested and what methodologies are to be used, and it should also include a general description of the tests themselves, including prerequisites, setup, execution, and a description of what to look for in the test results. The high-level outline is useful for administration, planning, and reporting, while the more detailed descriptions are meant to make the test process go smoothly. Often, the test plan has to account for the fact that time and budget constraints prohibit testing every component of a software system, and a risk analysis is one way of prioritizing the tests.

While not all testers like using test plans, test plans provide a number of benefits.

- They provide a written record of what is to be done.

286. #dsy255-BSI_risk-based-testing

- They allow project stakeholders to sign off on the intended testing effort. This helps ensure that the stakeholders agree with the elements of the plan and will support the test effort.
- Test plans provide a way to measure progress. This allows testers to determine whether they are on schedule, and also provides a concise way to report progress to the stakeholders.
- Due to time and budget constraints, it is often impossible to test all components of a software system. A test plan allows the analyst to succinctly record what the testing priorities are.

The reason that some testers dislike test plans is that they can constrain testers and prevent them from following up on observations and hunches that could lead to unexpected discoveries. This can be especially true in security testing, where the focus on negative requirements makes it desirable to investigate unexpected anomalies found during testing. However, a testing effort involving more than one person can easily deteriorate without a written plan. A good solution is to let testers deviate from the specific tests outlined in the test plan, but simply require that these changes be documented.

The information needed for test planning starts becoming available as soon as the software life cycle starts, but information continues arriving until the moment that the actual software artifact is ready for testing. In fact, the test process itself can generate information useful in the planning of further tests.

Therefore, test planning is an ongoing process. At its inception, a test plan is only a general outline of the intended test process, but more and more of the details are fleshed out as additional information becomes available.

For larger projects, the test plan is typically broken down into *test cycles.* This occurs for two reasons: first, the developing organization may modify software after problems are uncovered and then send the software back to be retested. Secondly, it is often inefficient for testing to begin only when development ends, so one component may be in testing while other components of the same system are still under development. In the first case, test cycles are created by the need to retest software that was already tested once before; in the second case, test cycles arise because the nature of the development effort implies that different modules will be tested at different times.

In creating the test plan, inter-component dependencies must be taken into account so that the potential need for retesting is minimized. In an ideal world, the testing organization would be able to actually specify the order in which components are tested, ensuring that each module is tested before other modules that might be dependent on it. In practice this is not so easy because the developing organization might be reluctant to change the planned sequence in which the components are to be developed. It may be that the testing organization can do no more than take these dependencies into account when creating the test plan.

Usually, a test plan also includes validation of the test environment and the test data. This is necessary because, for example, the test environment may fail to reflect the intended operational environment and crash the software, or the test data may be generated automatically and have an incorrect format.

Within each cycle, the test plan should map out the *test cases* that should be created during the test execution process. A test case typically includes information on the preconditions and postconditions of the test, information on how the test will be set up and how it will be torn down, and information about how the test results will be evaluated. The test case also defines a *test condition,* which is the actual state of affairs that is going to be tested; e.g., the tester creates the test condition in order to see how the software responds. The process of actually devising test conditions is a challenging one for security testing, and discussed in a separate section below.

The goal of test planning is to make the test process itself as automatic as possible, which not only makes the process go more smoothly but also makes it repeatable. Therefore the test plan should provide as much guidance as possible. In addition to the items already discussed, the test plan should specify what the tester should be looking for in each test, and if specific test preparations are required, then these should be included in the test plan too.

A typical security test plan might contain the following elements:

- Purpose
- Software Under Test Overview

    - Software and Components
    - Test Boundaries
    - Test Limitations
- Risk Analysis
    - Synopsis of Risk Analysis
- Test Strategy
    - Assumptions
    - Test Approach
    - Items Not To Be Tested
- Test Requirements
    - Functionality Test Requirements
    - Security Test Requirements
    - Installation/Configuration Test Requirements
    - Stress and Load Test Requirements
    - User Documentation
    - Personnel Requirements
    - Facility and Hardware Requirements
- Test Environment
- Test Case Specifications
    - Unique Test Identifier
    - Requirement Traceability (what requirement number from requirement document does test case validate)
    - Input Specifications
    - Output Specifications/Expected Results
    - Environmental Needs
    - Special Procedural Requirements
    - Dependencies Among Test Cases
- Test Automation and Testware
    - Testware Architecture
    - Test Tools
    - Testware
- Test Execution
    - Test Entry Criteria
    - QA Acceptance Tests
    - Regression Testing
    - Test Procedures

      Special requirements

      Procedure steps
    - Test Schedule
    - Test Exit Criteria
- Test Management Plan
- Definitions and Acronyms

Establishing the Test Environment

Testing requires the existence of a test environment. Establishing and managing a proper test environment is critical to the efficiency and effectiveness of a testing effort. For simple application programs, the test environment may consist of a single computer, but for enterprise-level software systems, the test environment may be much more complex, and the software may be closely coupled to the environment.

For security testing, it is often necessary for the tester to have more control over the environment than in other testing activities. This is because the tester must be able to examine and manipulate software/ environment interactions at a greater level of detail, in search of weaknesses that could be exploited by an attacker. The tester must also be able to control these interactions. The test environment should be isolated, especially if, for example, a test technique produces potentially destructive results during testing that might invalidate the results of any concurrent test activity.

Although it is not a good practice, it often happens that establishing the test environment is deferred until a week or so before the test execution begins. This may result in delays for starting execution, significant cost overruns, and encountering numerous problems at the onset of execution due to an inadequate or untested environment. The leader of the test stage will be responsible for ensuring that the architecture or technical environment team is beginning the process of establishing the environment during test planning so that it is ready to use for scripting.

The environment must be ready for the initial code migration two to four weeks prior to execution. Databases should also be populated via conversion programs or file load facilities. Once the environment is set up, a testing resource, in conjunction with a resource involved in setting up the environment, should execute portions of the test to ensure that the environment is set up correctly. This saves time up front by reducing the number of problems specifically related to the environment and ensures that all testers are productive at the onset of the test execution. In addition, this test serves as the inspection of the exit criteria of the environment setup.

Keep in mind that as the testing process progresses, environments from previous stages of testing will still be needed to support regression testing of defects.

Just prior to execution, the leader of a test stage will need to ensure that architecture and technical support personnel are allocated to support the environment and a support schedule is developed by areas of expertise (DBAs, network specialists, etc.), identifying a primary and secondary contact for each day of execution. This support schedule should be distributed to the test executors.

Setting up this operational environment early on prevents unforeseen testing delays that might be caused by nonessential setup errors. It also helps ensure a thorough understanding of risks associated with the operational environment.

This preparatory activity can best be understood by thinking of testing as an activity closely analogous to software development, involving its own requirements, specifications, support tools, version control, and so on. In a well-planned software development process, developers should not get stuck because of missing tools, missing information about how modules will work together, confusion about software versions, and so on, and the same is true of a well-planned test process.

Like risk analysis, test planning is a holistic process. It is also fractal in the sense that similar activities take place at different levels of abstraction: there is often a master test plan that outlines the entire test process, augmented by more detailed test plans for individual test stages, individual modules, and so on. For example, the master test plan for a web server might state that unit testing is carried out by developers, followed by module testing performed by test engineers, integration testing in a straw-man environment, system testing in a simulated real environment, and stress testing with specialized tools (a real test plan generally includes more phases than this). Each of these test phases might have its own test plan, and for each phase there may be a test plan for each artifact that is being tested.

Relevant Metrics

Gerald Weinberg once stated, ”It’s possible to meet any goal on a software project so long as the quality standard is not fixed.” It’s true. It’s simple to make a project appear as if it’s running on schedule when you don’t have to meet any quality standard. Every status report needs to show objective measures of progress, such as how many tests have been executed and how many are passing. It should also show a cumulative graph of the number of defects reported and the number fixed. Without this information, it is meaningless to claim that a project is on schedule.

Both the quantity and quality of testing needs to be measured to help make a quantifiable assessment about how well the software has been tested. Unfortunately, there are no industry-standard metrics for measuring test quantity and test quality, although several published papers have proposed such measures. In general, test metrics are used as a means of determining when to stop testing and when to release the software to the customer.

The test criteria to be considered and the rationale behind using these criteria are described below.

- Test requirements criteria**:** One of the purposes of testing is to demonstrate that the software functions as specified by the requirements. Thus every software requirement must be tested by at least one corresponding test case. By having traceability from test cases to functional requirements, one can determine the percentage of requirements tested to the total number of requirements. Thus, *test requirement metric* is defined as the ratio of requirements tested to the total number of requirements.
- Test coverage criteria**:** Testing should attempt to exercise as many ”parts” of the software as possible, since code that is not exercised can potentially hide faults. Exercising code comprehensively involves covering *all* the execution paths through the software. Unfortunately, the number of such paths in even a small program can be astronomical, and hence it is not practical to execute all these paths. The next best thing is to exercise as many statements, branches, and conditions as possible. Thus, we can define the test coverage criteria such as statement coverage, branch coverage, etc. Standard off-the-shelf coverage tools can be used to monitor coverage levels as testing progresses.
- Test case metric**:** This metric is useful when it is plotted with respect to time. Essentially, such a plot will show the total number of test cases created, how many have been exercised, how many test cases have passed, and how many have failed over time. This project-level metric provides a snapshot of the progress of testing#where the project is today and what direction it is likely to take tomorrow.
- Defect metrics: This metric is also very useful when plotted with respect to time. It is more informative to plot cumulative defects found since the beginning of testing to the date of the report. The slope of this curve provides important information. If the slope of the curve is rising sharply, large numbers of defects are being discovered rapidly, and certainly testing efforts must be maintained, if not intensified. In contrast, if the slope is leveling off, it might mean fewer failures are being found. It might also mean that the quality of testing has decreased.

  Ultimately, the effectiveness of testing can only be deduced when data on defects is collected in actual operational use after the software is released. It is recommended that defects found during actual operational use be measured consistently at two standard points after delivery to the user(s), such as three and six months. (This will clearly depend on the release schedule. For example, a six month sampling period will not be appropriate for software with a four month release schedule.)
- Test case quality metric: Test cases ”fail” when the application under test produces a result other than what is expected by the test case. This can happen due to several reasons, one being a true defect in the application. Other reasons could be a defect in the test case itself, a change in the application, or a change in the environment of the test case. A test case can be defined to be of high quality whenever it finds a true defect in the application. Test case quality metric can be defined as the ratio of test case failures resulting in a defect to the total number of test case failures.

The most important of these metrics are defect metrics. Defect metrics must be collected and carefully analyzed in the course of the project. These are very important data. Some of these metrics may require that the problem tracking system be modified.

Information to track about each defect includes

- date found
- date resolved
- status (open: need attention by owner; resolved: decision or fix made, needs verification by QA; concluded: resolution verified by QA)
- resolution (fixed, deferred, cannot reproduce, rejected - not a bug, duplicate)
- brief description
- detailed description (steps to reproduce, expected results, actual results, notes)
- priority
- severity
- waiver status (used after code freeze)
- reported by
- assigned to
- found in (build version)
- fixed in (build version)
- product area
- module(s)
- error type (from error taxonomy)
- fix effort (in hours)
- how found (test case ###, exploratory)
- comments
- audit trail of any changes to the problem report

The following is a brief description of the priority and severity fields in a defect report.

Priority: This is a measure of the probability that the failure mode can occur in the field during typical usage, such as a scale from 1 (most damage) to 5 (least damage).

Severity: This denotes the absolute severity of the failure mode in question, regardless of probability of occurrence. The scale could be as follows:

1. loss of data, system completely unusable (can't use system, user is blocked)
2. loss of functionality with no workaround
3. loss of functionality with a workaround
4. partial loss of functionality
5. cosmetic or trivial

Reporting

All time-oriented metrics should be measured on a daily scale. These metrics should be automated so they are updated on a daily basis. The reports should be available for the whole system and on the level of individual product areas.

These are the major reports to produce. Others may be useful, depending on the circumstances:

- total open, by priority
- stop-ship problems (as indicated by waiver field)
- find rate, by severity (daily and 15-day rolling average)
- resolution type versus severity for all resolved problems (total and last 15 days)
- unreviewed problems, by priority
- unverified resolutions (a.k.a. QA backlog)
- who found (by function)

- reopen rate, over time (number of problems with resolutions that were rejected by QA, and thus reopened rather than concluded)

Collection Protocol

The problem database should be maintained primarily by the test team. All problems found by anyone during and after functional testing must be logged in the database. Encourage the developers to submit problems to testers for entry into the database, since the alternative is that many problem reports won't be logged at all.

This protocol can be enforced easily by requiring that all changes made after code freeze are associated with a problem number. Prior to each build, the change list is then compared to the problem tracking system.

Problems found by customer engineering, and ultimately the customer, are also important to track.

How To Use the Metrics

Defect metrics are vital to the successful management of a high assurance test project. A complete treatment of the subject is beyond the scope of this report. However, here are a few important points to keep in mind:

- Defect metrics should be used by the test manager and the project manager, on more or less a daily basis, to review the status of the product, scan for risks, and guide the testing effort toward areas of greatest risk.
- The product should ship with 0 open problem reports, and 0 resolved but not verified. All problem reports should be resolved and verified.
- We do not recommend a find rate requirement (e.g., no defects found for a week) for shipping the product, since such requirements are arbitrary and subject to accidental abuse. We do recommend reviewing each and every problem found in the final third or quarter of the project and using that information to question the efficacy of the test effort and to re-analyze technical risk. This is far more useful than an aggregate find rate at apprising management of the product's readiness to ship.
- While we don't advocate a find rate requirement for shipping, a 15-day rolling average defect find rate that drops substantially below the highest fifteen day rolling average find rate for important bugs (say, the top two severity or priority categories) in the project should trigger inquiry into whether a new release is needed or whether new tests or test techniques should be employed. Keeping the important problem find rate as high as possible throughout the project is an important goal of the test team.
- The problem reopen rate is an indicator of the competence of the testers, the health of the relationship between testing and development, the testability of the product, and the thoroughness of developers in investigating and resolving problems. In general, the reopen rate should start low (less than 1 out of 10 resolutions rejected) and get steadily lower over the course of the project.

Case Study

Although it is strongly recommended that an organization not rely exclusively on security test activities to build security into a system, security testing, when coupled with other security activities performed throughout the SDLC, can be very effective in validating design assumptions, discovering vulnerabilities associated with the application environment, and identifying implementation issues that may lead to security vulnerabilities.

For example, an organization had assembled a large software development team to build a high-profile Internet-based gaming system. The gaming system was planned to augment an existing, government-sponsored, paper-based gaming system. Understanding the broad and potentially significant security implications relating to the system, the development organization made every effort to design security into the product. A security architect was involved with the development effort from initial requirement generation through system delivery. Security activities were conducted throughout the SDLC to ensure that security was built into the system. These included the following:

- Security-based requirements were developed.

- Security-based risk assessments to identify areas of greatest risk to the business and the technology platform were completed.
- Findings from the risk assessments were addressed in the security architecture and implementation.
- Security-based design and architecture reviews were conducted.
- Security training was provided to developers.
- Code reviews were conducted on security-critical components.

Despite these efforts, an issue associated with the input validation component was identified during system-level security testing. Although input validation was engineered into the overall design and the component had been previously approved in both design and code reviews, there was an issue. The source of the problem was later identified to be associated with the build process. An incorrectly functioning and previously rejected input validation component had made its way into the final build. Had it not been for the final system-level security test activity, the system would have been deployed with the faulty input validation mechanism.

Acknowledgments

Some additional material in this document was provided by Girish Janardhanudu and Carol Lemberg. Some other material is taken from internal Cigital, Inc. documents whose authorship cannot be definitely attributed but includes James Bach, Brian Marick, Kamesh Pammeraju, and Danny Faught.

Glossary

acceptance testing

Formal testing conducted to enable a user, customer, or other authorized entity to determine whether to accept a system or component. [IEEE 90[430]]

ad hoc testing

Testing carried out using no recognized test case design technique. [BS-7925[431]]

authentication

The process of confirming the correctness of the claimed identity. [SANS 03[432]]

black box testing

Testing that is based on an analysis of the specification of the component without reference to its internal workings. [BS-7925[433]]

buffer overflow

A buffer overflow occurs when a program or process tries to store more data in a data storage area than it was intended to hold. Since buffers are created to contain a finite amount of data, the extra information—which has to go somewhere—can overflow into the runtime stack, which contains control information such as function return addresses and error handlers.

buffer overflow attack

See stack smashing.

bug

See fault.

430. #dsy255-BSI_refs
431. #dsy255-BSI_refs
432. #dsy255-BSI_refs
433. #dsy255-BSI_refs

capture/replay tool

A test tool that records test input as it is sent to the software under test. The input cases stored can then be used to reproduce the test at a later time. [BS-7925[434]]

compatibility testing

Testing whether the system is compatible with other systems with which it should communicate. [BS-7925[435]]

component

A minimal software item for which a separate specification is available. [BS-7925[436]]

conformance testing

The process of testing that an implementation conforms to the specification on which it is based. [BS-7925[437]]

cookie

Data exchanged between an HTTP server and a browser (a client of the server) to store state information on the client side and retrieve it later for server use. An HTTP server, when sending data to a client, may send along a cookie, which the client retains after the HTTP connection closes. A server can use this mechanism to maintain persistent client-side state information for HTTP-based applications, retrieving the state information in later connections. [SANS 03[438]]

correctness

The degree to which software conforms to its specification. [BS-7925[439]]

cryptographic attack

A technique for successfully undermining an encryption scheme.

cryptography

Cryptography garbles a message in such a way that anyone who intercepts the message cannot understand it. [SANS 03[440]]

domain

The set from which values are selected. [BS-7925[441]]

domain testing

Testing with test cases based on the specification of input values accepted by a software component. [Beizer 90[442]]

dynamic analysis

The process of evaluating a system or component based on its behavior during execution. [IEEE 90[443]]

Dynamic Link Library (DLL)

434. #dsy255-BSI_refs
435. #dsy255-BSI_refs
436. #dsy255-BSI_refs
437. #dsy255-BSI_refs
438. #dsy255-BSI_refs
439. #dsy255-BSI_refs
440. #dsy255-BSI_refs
441. #dsy255-BSI_refs
442. #dsy255-BSI_refs
443. #dsy255-BSI_refs

A collection of small programs, any of which can be called when needed by a larger program that is running in the computer. Small programs that enable larger programs to communicate with a specific device such as a printer or scanner are often packaged as DLL programs (usually referred to as DLL files). [SANS 03[444]]

encryption

Cryptographic transformation of data (called ”plaintext”) into a form (called ”cipher text”) that conceals the data’s original meaning to prevent it from being known or used. [SANS 03[445]]

failure

The inability of a system or component to perform its required functions within specified performance requirements. [IEEE 90[446]]

fault

A manifestation of an error in software. A fault, if encountered, may cause a failure. [RTCA 92[447]]

Hypertext Transfer Protocol (HTTP)

The protocol in the Internet Protocol (IP) family used to transport hypertext documents across an internet. [SANS 03[448]]

integration testing

Testing performed to expose faults in the interfaces and in the interaction between integrated components. [BS-7925[449]]

interface testing

Integration testing in which the interfaces between system components are tested. [BS-7925[450]]

isolation testing

Component testing of individual components in isolation from surrounding components, with surrounding components being simulated by stubs. [BS-7925[451]]

kernel

The essential center of a computer operating system, the core that provides basic services for all other parts of the operating system. A synonym is nucleus. A kernel can be contrasted with a shell, the outermost part of an operating system that interacts with user commands. Kernel and shell are terms used more frequently in UNIX and some other operating systems than in IBM mainframe systems. [SANS 03[452]]

National Institute of Standards and Technology (NIST)

A unit of the U.S. Commerce Department. Formerly known as the National Bureau of Standards, NIST promotes and maintains measurement standards. It also has active programs for encouraging and helping industry and science to develop and use these standards. [SANS 03[453]]

negative requirements

444. #dsy255-BSI_refs
445. #dsy255-BSI_refs
446. #dsy255-BSI_refs
447. #dsy255-BSI_refs
448. #dsy255-BSI_refs
449. #dsy255-BSI_refs
450. #dsy255-BSI_refs
451. #dsy255-BSI_refs
452. #dsy255-BSI_refs
453. #dsy255-BSI_refs

Requirements that state what software should not do.

operational testing

Testing conducted to evaluate a system or component in its operational environment. [IEEE 90[454]]

port

A port is nothing more than an integer that uniquely identifies an endpoint of a communication stream. Only one process per machine can listen on the same port number. [SANS 03[455]]

precondition

Environmental and state conditions that must be fulfilled before the component can be executed with a particular input value.

protocol

A formal specification for communicating; the special set of rules that end points in a telecommunication connection use when they communicate. Protocols exist at several levels in a telecommunication connection. [SANS 03[456]]

pseudorandom

Appearing to be random, when actually generated according to a predictable algorithm or drawn from a prearranged sequence.

race condition

A race condition exploits the small window of time between a security control being applied and the service being used. [SANS 03[457]]

registry

The registry in Windows operating systems is the central set of settings and information required to run the Windows computer. [SANS 03[458]]

regression testing

Retesting of a previously tested program following modification to ensure that faults have not been introduced or uncovered as a result of the changes made. [BS-7925[459]]

requirement

A capability that must be met or possessed by the system/software (requirements may be functional or non-functional). [BS-7925[460]]

requirements-based testing

Designing tests based on objectives derived from requirements for the software component (e.g., tests that exercise specific functions or probe the non-functional constraints such as performance or security). [BS-7925[461]]

reverse engineering

454. #dsy255-BSI_refs
455. #dsy255-BSI_refs
456. #dsy255-BSI_refs
457. #dsy255-BSI_refs
458. #dsy255-BSI_refs
459. #dsy255-BSI_refs
460. #dsy255-BSI_refs
461. #dsy255-BSI_refs

Acquiring sensitive data by disassembling and analyzing the design of a system component [SANS 03[462]]; acquiring knowledge of a binary program's algorithms or data structures.

risk assessment

The process by which risks are identified and the impact of those risks is determined. [SANS 03[463]]

security policy

A set of rules and practices that specify or regulate how a system or organization provides security services to protect sensitive and critical system resources. [SANS 03[464]3]

server

A system entity that provides a service in response to requests from other system entities called clients. [SANS 03[465]]

session

A virtual connection between two hosts by which network traffic is passed. [SANS 03[466]]

socket

The socket tells a host's IP stack where to plug in a data stream so that it connects to the right application. [SANS 03[467]]

software

Computer programs (which are stored in and executed by computer hardware) and associated data (which also is stored in the hardware) that may be dynamically written or modified during execution. [SANS 03[468]]

specification

A description, in any suitable form, of requirements. [BS-7925[469]]

specification testing

An approach to testing wherein the testing is restricted to verifying that the system/software meets the specification. [BS-7925[470]]

SQL Injection

SQL injection is a type of input validation attack specific to database-driven applications where SQL code is inserted into application queries to manipulate the database. [SANS 03[471]]

stack smashing

The technique of using a buffer overflow to trick a computer into executing arbitrary code. [SANS 03[472]]

state transition

462. #dsy255-BSI_refs
463. #dsy255-BSI_refs
464. #dsy255-BSI_refs
465. #dsy255-BSI_refs
466. #dsy255-BSI_refs
467. #dsy255-BSI_refs
468. #dsy255-BSI_refs
469. #dsy255-BSI_refs
470. #dsy255-BSI_refs
471. #dsy255-BSI_refs
472. #dsy255-BSI_refs

A transition between two allowable states of a system or component. [BS-7925[473]]

state transition testing

A test case design technique in which test cases are designed to execute state transitions. [BS-7925[474]]

static analysis

Analysis of a program carried out without executing the program. [BS-7925[475]]

static analyzer

A tool that carries out static analysis. [BS-7925[476]]

stress testing

Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements. [IEEE 90[477]]

stub

A skeletal or special-purpose implementation of a software module used to develop or test a component that calls or is otherwise dependent on it. [IEEE 90[478]].

syntax testing

A test case design technique for a component or system in which test case design is based on the syntax of the input. [BS-7925[479]]

system testing

The process of testing an integrated system to verify that it meets specified requirements. [Hetzel 88[480]]

test automation

The use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions.

test case

A set of inputs, execution preconditions, and expected outcomes developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. [IEEE 90[481]]

test suite

A collection of one or more test cases for the software under test. [BS-7925[482]]

test driver

A program or test tool used to execute software against a test suite. [BS-7925[483]]

test environment

473. #dsy255-BSI_refs
474. #dsy255-BSI_refs
475. #dsy255-BSI_refs
476. #dsy255-BSI_refs
477. #dsy255-BSI_refs
478. #dsy255-BSI_refs
479. #dsy255-BSI_refs
480. #dsy255-BSI_refs
481. #dsy255-BSI_refs
482. #dsy255-BSI_refs
483. #dsy255-BSI_refs

A description of the hardware and software environment in which tests will be run and any other software with which the software under test interacts when under test, including stubs and test drivers. [BS-7925[484]]

test plan

A record of the test planning process detailing the degree of tester independence, the test environment, the test case design techniques and test measurement techniques to be used, and the rationale for their choice. [BS-7925[485]]

testware

Software associated with carrying out tests, such as test drivers, stubs, and software needed to set up and tear down test cases.

vulnerability

A defect or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy. [SANS 03[486]]

web server

A software process that runs on a host computer connected to the Internet to respond to HTTP requests for documents from client web browsers.

References

[Beizer 90]	Beizer, Boris. *Software Testing Techniques*, Chapter 10. New York, NY: van Nostrand Reinhold, 1990 (ISBN 0-442-20672-0).
[Beizer 95]	Beizer, Boris. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. New York, NY: John Wiley & Sons, 1995.
[Binder 99]	Binder, R. V. *Testing Object-Oriented Systems: Models, Patterns, and Tools* (Addison-Wesley Object Technology Series). Boston, MA: Addison-Wesley Professional, 1999.
[Black 02]	Black, Rex. *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing, 2nd ed*. New York, NY: John Wiley & Sons, 2002.
[BS 7925]	British Computer Society. *Glossary of terms used in software testing*[488] (BS 7925-1).
[Campbell 03]	Campbell, Katherine; Gordon, Lawrence A.; Loeb, Martin P.; & Zhou, Lei. "The Economic Cost of Publicly Announced Information Security Breaches: Empirical Evidence From the Stock Market." *Journal of Computer Security 11*, 3 (2003).
[DeVale 99]	DeVale, J.; Koopman, P.; & Guttendorf, D. "The Ballista Software Robustness Testing Service[489]," 33-42. 16th International Conference on Testing

484. #dsy255-BSI_refs
485. #dsy255-BSI_refs
486. #dsy255-BSI_refs

	Computer Software. Washington, D.C., June 14-18, 1999.
[Dijkstra 70]	Dijkstra, E. W. "Structured Programming." *Software Engineering Techniques*. Edited by J. N. Buxton and B. Randall. Brussels, Belgium: NATO Scientific Affairs Division, 1970, pp. 84-88.
[Du 00]	Du, W. & Mathur, A. P. "Testing for Software Vulnerability Using Environment Perturbation," 603-612. *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000)*, Workshop On Dependability Versus Malicious Faults. New York, NY, June 25-28, 2000. Los Alamitos, CA: IEEE Computer Society Press, 2000.
[Du 98]	Du, W. & Mathur, A. P. *Vulnerability Testing of Software System Using Fault Injection* (COAST technical report). West Lafayette, IN: Purdue University, 1998.
[Dustin 01]	Dustin, Elfriede; Rashka; Jeff; McDiarmid, Douglas; & Nielson, Jakob. *Quality Web Systems: Performance, Security, and Usability*. Boston, MA: Addison Wesley Professional, 2001.
[Dustin 99]	Dustin, E.; Rashka, J.; & Paul, J. *Automated Software Testing*. Boston, MA: Addison Wesley Professional, 1999.
[Faust 04]	Faust, S. "Web Application Testing with SPI Fuzzer." SPI Dynamics Whitepaper, 2004.
[Fewster 99]	Fewster, Mark & Graham, Doroty. *Software Test Automation*. Boston, MA: Addison-Wesley Professional, 1999.
[Fink 97]	Fink, G. & Bishop, M. "Property-Based Testing: A New Approach to Testing for Assurance." ACM SIGSOFT *Software Engineering Notes 22*, 4 (July 1997): 74-80.
[Friedman 95]	Friedman, Michael A. & Voas, Jeffrey M. *Software Assessment: Reliability, Safety, Testability*. Wiley InterScience, 1995.
[Ghosh 98]	Ghosh, Anup K.; O'Connor, Tom; & McGraw, Gary. "An Automated Approach for Identifying Potential Vulnerabilities in Software," 104-114. *Proceedings of the 1998 IEEE Symposium on Security and Privacy*. Oakland, California, May 3-6, 1998. Los Alamitos, CA: IEEE Computer Society Press, 1998.
[Graff 03]	Graff, Mark G. & Van Wyk, Kenneth R. *Secure Coding: Principles and Practices*. Sebastopol, CA: O'Reilly, 2003 (ISBN: 0596002424).
[Grance 02]	Grance, T.; Myers, M.; & Stevens, M. *Guide to Selecting Information Technology Security Products* (NIST Special Publication 800-36[490]), 2002.

[Grance 04]	Grance, T.; Myers, M.; & Stevens, M. Security Considerations in the Information System Development Life Cycle (NIST Special Publication 800-64[491]), 2004.
[Guttman 95]	Guttman, Barbara & Roback, Edward. *An Introduction to Computer Security*[492]. Gaithersburg, MD: U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, 1995.
[Hetzel 88]	Hetzel, William C. *The Complete Guide to Software Testing, 2nd ed*. Wellesley, MA: QED Information Sciences, 1988.
[Hoglund 04]	Hoglund, Greg & McGraw, Gary. *Exploiting Software: How to Break Code*. Boston, MA: Addison-Wesley, 2004.
[Howard 02]	Howard, Michael & LeBlanc, David. *Writing Secure Code, 2nd ed*. Redmond, WA: Microsoft Press, 2002.
[Howard 06]	Howard, Michael & Lipner, Steve. *The Security Development Lifecycle*. Redmond, WA: Microsoft Press, 2006, ISBN 0735622142.
[Hsueh 97]	Hsueh, Mei-Chen; Tsai, Timothy K.; & Lyer, Ravishankar K. "Fault Injection Techniques and Tools." *Computer 30*, 4 (April 1997): 75-82.
[Hunt 99]	Hunt, G. & Brubacher, D. "Detours: Binary Interception of Win32 Functions[494]." USENIX Technical Program - Windows NT Symposium 99. Seattle, Washington, July 12-15, 1999.
[IEEE 90]	IEEE. *IEEE Standard Glossary of Software Engineering Terminology* (IEEE Std 610.12-1990). Los Alamitos, CA: IEEE Computer Society Press, 1990.
[Jones 94]	Jones, Capers. *Assessment and Control of Software Risks*. Englewood Cliffs, NJ: Yourdon Press, 1994.
[Kaksonen 02]	Kaksonen, R. "A Functional Method for Assessing Protocol Implementation Security." Technical Research Centre of Finland, VTT Publications 48, 2002.
[Kaner 99]	Kaner, Cem; Falk, Jack; & Nguyen, Hung Quoc. *Testing Computer Software, 2nd ed*. New York, NY: John Wiley & Sons, 1999.
[Leveson 95]	Leveson, N. O. *Safeware: System Safety and Computers*. Reading, MA: Addison-Wesley, 1995.
[Marick 94]	Marick, Brian. *The Craft of Software Testing: Subsystems Testing Including Object-Based and Object-Oriented Testing*. Upper Saddle River, NJ: Prentice Hall PTR, 1994.

[McGraw 04a]	McGraw, Gary & Potter, Bruce. "Software Security Testing." *IEEE Security and Privacy 2*, 5 (Sept.-Oct. 2004): 81-85.
[McGraw 04b]	McGraw, Gary. "Application Security Testing Tools: Worth the Money?[496]" *Network Magazine*, November 1, 2004. (2004).
[Miller 90]	Miller, Barton P.; Fredriksen, Lars; & So, Bryan. "An empirical study of the reliability of UNIX utilities." *Communications of the ACM 33*, 12 (December 1990): 32-44.
[Miller 95]	Miller, B.; Koski, D.; Lee, C.; Maganty, V.; Murthy, R.; Natarajan, A.; & Steidl, J. *Fuzz Revisited: A Re-Examination of the Reliability of Unix Utilities and Services*. Technical report, Computer Sciences Department, University of Wisconsin, 1995.
[NIST 02a]	NIST. *The Economic Impacts of Inadequate Infrastructure for Software Testing* (Planning Report 02-3[497]). Gaithersburg, MD: National Institute of Standards and Technology, 2002.
[Ricca 01]	Ricca, F. & Tonella, P. "Analysis and Testing of Web Applications," 25–34. *Proceedings of the 23rd IEEE International Conference on Software Engineering*. Toronto, Ontario, Canada, May 2001. Los Alamitos, CA: IEEE Computer Society Press, 2001.
[Richardson 08]	Richardson, Robert. *2008 CSI/FBI Computer Crime and Security Survey*. San Francisco, CA: Computer Security Institute, 2008.
[RTCA 92]	RTCA, Inc. *DO-178B, Software Considerations in Airborne Systems and Equipment Certification*. Issued in the U.S. by RTCA, Inc. (document RTCA SC167/DO-178B) and in Europe by the European Organization for Civil Aviation Electronics (EUROCAE document ED-12B), December 1992.
[Rukhin 01]	Rukhin, Andrew; Soto, Juan; Nechvatal, James; Smid, Miles; Barker, Elaine; Leigh, Stefan; Levenson, Mark; Vangel, Mark; Banks, David; Heckert, Alan; Dray, James; & Vo, San. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications* (NIST Special Publication 800-22)[498], 2001.
[SANS 03]	The SANS Institute. *SANS Glossary of Terms Used in Security and Intrusion Detection*[499] (2003).
[Schneier 00b]	Schneier, B. *Secrets and Lies: Digital Security in a Networked World*. New York: John Wiley & Sons, 2000.

[SPI 02]	SPI Dynamics. "SQL Injection: Are Your Web Applications Vulnerable?" (white paper). Atlanta, GA: SPI Dynamics, 2002.
[SPI 03]	SPI Dynamics. "Web Application Security Assessment" (white paper). Atlanta, GA: SPI Dynamics, 2003.
[Verdon 04]	Verdon, Denis & McGraw, Gary. "Risk Analysis in Software Design." *IEEE Security & Privacy 2*, 4 (July-Aug. 2004): 79-84.
[Viega 03]	Viega, John & Messier, Matt. *Secure Programming Cookbook for C and C++*. Sebastopol, CA: O'Reilly, 2003 (ISBN: 0596003943).
[Voas 95]	Voas, Jeffrey M. & Miller, Keith W. "Examining Fault-Tolerance Using Unlikely Inputs: Turning the Test Distribution Up-Side Down," 3-11. *Proceedings of the Tenth Annual Conference on Computer Assurance*. Gaithersburg, Maryland, June 25-29, 1995. Los Alamitos, CA: IEEE Computer Society Press, 1995.
[Voas 98]	Voas, Jeffrey M. & McGraw, Gary. *Software Fault Injection: Inoculating Programs Against Errors*, 47-48. New York, NY: John Wiley & Sons, 1998.
[Wack 03]	Wack, J.; Tracey, M.; & Souppaya, M. *Guideline on Network Security Testing* (NIST Special Publication 800-42[500]), 2003.
[Whalen 05]	Whalen, Sean; Bishop, Matt; & Engle, Sophie. "Protocol Vulnerability Analysis[501]," UC Davis Department of Computer Science Technical Report CSE-2005-04, May 2005.
[Whittaker 02]	Whittaker, J. A. *How to Break Software*. Reading MA: Addison Wesley, 2002.
[Whittaker 03]	Whittaker, J. A. & Thompson, H. H. *How to Break Software Security*. Reading MA: Addison Wesley, 2003.
[Wysopal 03]	Wysopal, Chris; Nelson, Lucas; Zovi, Dino Dai; & Dustin, Elfriede. *The Art of Software Security Testing*. Upper Saddle River, NJ: Pearson Education, Inc.

Cigital, Inc. Copyright

1. mailto:copyright@cigital.com